Classification of reviews based on text

Riccardi Alessio

I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work. I understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. This assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study.

1 Introduction

The development of this project consists in the use of a deep learning technique to carry out the classification of reviews based on the text. In particular, a feed forward neural network was used, whose input consists of several features extracted from the text of the reviews. Finally, several metrics were used to evaluate the result obtained from the machine learning model.
The project is written in python with the use of Apache Spark which is a multi-language engine for executing data engineering, data science, and machine learning on single-node machines or clusters.
This report will present how the project scale with the size of the data, dataset used, how the data is organized, the pre-processing techniques used on the text, the features extraction techniques use, the model used for classification and the various classification results obtained.

2 Scalability

The solution that will be presented in the following chapters is able to scale with the size of the data thanks to the use of Spark. In particular, the Spark NLP and Spark MLlib (later called Spark ML) libraries were used which are built on Apache Spark. Both provide many useful classes in the field of NLP and machine learning, making them particularly efficient in the field of distributed computing. Furthermore, they are able to scale when dealing with big data. In order to start a Spark NLP Session you need to use the *sparknlp.start()* instruction.
Within the project, the main steps (preprocessing, feature extraction and model

training) were carried out via a Pipeline made up of different stages (one for each-preprocessing and feature extraction step) and via multilayer perceptron model both available in the Spark ML library. This allows you to execute the project in a reasonable time and with a good amount of input data.

3 Dataset

The dataset (version 4) used in this project is a subset of Yelp's businesses, reviews and user data. In this dataset one can find information about businesses across 8 metropolitan areas in the USA and CANADA. There are 5 JSON files inside the dataset. In this paper we are mainly interested in the yelp-academic-dataset-review.json file, which contains the text of the reviews and the related ratings that will be the subject of analysis.

4 How Data is organized

Once the downloaded dataset was unzipped, the json file being analyzed was read using spark in order to create a dataframe. In particular, a Spark dataframe was created so that there was compatibility with the Spark ML and Spark NLP libraries used for the development of the project.
From the dataframe created, it was decided to consider only two columns, the one containing the text and the one containing the ratings. In fact, this is the only information that will be necessary in order to carry out the classification via neural network.
From the original dataset, 5 dataframe were created representing the five ratings for which a prediction needs to be made. This step was necessary in order to obtain a balance between the various classes, thus avoiding the use of oversampling or undersampling operations. Finally, these dataframe were merged into a single dataset to which a shuffling operation was applied in order to mix the rows. In fact, after the merge operation, all the rows with the same rating appear close.

5 Pre-Processing

In this phase, a pipeline with different stages was defined, each representing a preprocessing operation on the text. In particular, the main pre-processing steps were performed such as tokenization, normalization and lemmatization of the text and the removal of stop words.
For what concern the pipeline its definition is possible thanks to the use of the Spark ML library which also provides methods for both feature extractions and feature transformations. Furthermore, annotator models of the Spark NLP library were used to carry out the pre-processing operations. Annotator Models are Spark models or transformers, meaning they have a transform(data) function. This function takes as input a dataframe to which it adds a new column containing the result of the current annotation. All transformers are additive, meaning they append to current data, never replace or delete previous information.
The first step of pre-processing involves the use of the Document Assembler annotator

which prepares the data in a format that can be processed by Spark NLP. This is the entry point for every Spark NLP pipeline. Then the SentenceDetector annotator was used which is able to detect sentence boundaries using regular expressions. An example of a regular expression used is that relating to punctuation. However, you can also specify a custom boundary if necessary. After that, the Tokenize annotator was used to tokenize the raw text in the sentence type column into tokenized sentences. Lemmatization was applied to the result of the Tokenize annotator. In particular, the LemmatizerModel annotator was used which allows you to load the default pretrained model called *lemma-antbnc*. The Lemmatizer allows you to find lemmas out of words with the objective of returning a base dictionary word. Retrieves the significant part of a word. Finally, the text normalization step was carried out using an annotator called Normalizer. This annotator cleans out tokens. Requires stems, hence tokens and removes all dirty characters from text following a regex pattern and transforms words based on a provided dictionary. Since no specific dictionary was passed, only the alphabetical letters are retained in the text.
Before removing the stopwords, another annotator called Finisher was used. The latter serves to convert the annotation results into a format that is easier to use. In particular, it is useful to extract the results from Spark NLP Pipelines. The Finisher outputs annotation(s) values into String. This step was necessary since the StopWordsRemover class of the *pyspark.ml.feature* library was used which requires string values and not annotator values as input. The *loadDefaultStopWords* method of this class was used in order to set the default stopwords for the English language.

6 Features Extraction

As regards the features to be used as input for the machine learning model, it was decided to use a simple TF.IDF in order to try to find which were the most important words that could characterize the text for a given set of reviews having the same rating. As its name implies, TF-IDF vectorizes/scores a word by multiplying the word's Term Frequency (TF) with the Inverse Document Frequency (IDF). Term Frequency of a term or word is the number of times the term appears in a document compared to the total number of words in the document. IDF of a term reflects the proportion of documents in the corpus that contain the term. Words unique to a small percentage of documents receive higher importance values than words common across all documents. To compute the term frequency, the CountVectorizer class was used which extracts a vocabulary from document collections and generates a CountVectorizerModel. This class was given as input the column representing the filtered tokens (the output of the StopWordRemover class), the vocabulary to extract (6500 in this case) and a minDF value set to 2, which specifies the minimum number of documents in which the term must appear. Instead, to produce the IDF score, the IDF class was used, which starting from the column produced by CountVectorizer, compute the inverse document frequency for the terms. The column resulting from this computation was named features and was passed as the input layer of the multilayer perceptron.
Both the CountVectorizer class and the IDF class are available in the Spark ML library.

7 Model

As mentioned above, a feed-forward neural network was used as the model for classification. In particular, a multilayer perceptron available from the Spark ML library was used. Multilayer perceptron classifier (MLPC) consists of multiple layers of nodes. Each layer is fully connected to the next layer in the network. Nodes in the input layer represent the input data (features extracted using TF.IDF in our case). All other nodes maps inputs to the outputs by performing linear combination of the inputs with the node's weights w and bias b and applying an activation function. Nodes in intermediate layers use sigmoid function:

$$f(x) = \frac{1}{1 + e^{-x}}$$

This function takes any real value as input and outputs values in the range of 0 to 1. The larger the input (more positive), the closer the output value will be to 1.0, whereas the smaller the input (more negative), the closer the output will be to 0.0, as shown below.
Nodes in the output layer use softmax function:

$$f(z_i) = \frac{e_i^z}{\sum_{k=1}^{N} e_k^z}$$

Softmax is an activation function that scales numbers/logits into probabilities. The output of a Softmax is a vector v with probabilities of each possible outcome. The probabilities in vector v sums to one for all possible outcomes or classes. In the case of output nodes, these correspond to the number of classes (in this case the 5 values for the ratings). Furthermore, MLPC employes backpropagation for learning the model, logistic loss function for optimization and L-BFGS as optimization routine. L-BGFS is an optimization algorithm in the family of quasi-Newton methods to solve the optimization problems of the form $min_{w \in R^d} f(w)$. The L-BFGS method approximates the objective function locally as a quadratic without evaluating the second partial derivatives of the objective function to construct the Hessian matrix. The Hessian matrix is approximated by previous gradient evaluations, so there is no vertical scalability issue (the number of training features) when computing the Hessian matrix explicitly in Newton's method. As a result, L-BFGS often achieves more rapid convergence compared with other first-order optimization.

7.1 Training and Validation

Before carrying out the training, the value of the ratings had to be decreased by 1, taking them from the interval [1.0, 5.0] to the interval [0.0, 4.0]. This is because the multilayer perceptron in the case of multi-class only accepts labels in the range [0, maxvalue). After that, the dataset was split into training, validation and test sets with a percentage of [0.7, 0.3, 0.15] using the *randomSplit* function.
It was decided to also use the validation set because although different models were not used, we tried to do some hyperparameter tuning, ending up choosing the best

result produced by the model on which to use the test set. We tried performing different training sessions with different values of batch size, learning rate, modifying the number of neurons in the hidden layers and trying to add more than one hidden layer. The number of iterations performed by the neural network is equal to 80, while the number of features extracted via TF.IDF is always used as the input layer.
In the table 1 it is possible to observe the precision, recall, f1-score and accuracy results obtained on the training and validation set. In particular, it can be seen that by increasing the number of hidden layers or decreasing the number of neurons in the single hidden layer, performance tends to decrease. In the next section, the results of the classification metrics obtained from the application of the model on the test set containing unseen data will be analyzed.

	learning rate	batch size	layers	val precision	val recall	val accuracy	val f1-score	train precision	train recall	train accuracy	train f1-score
1	0.0001	32	1 hidden layer with 32 neurons	0.79	0.79	0.79	0.79	0.79	0.79	0.79	0.79
2	0.0001	64	1 hidden layer with 32 neurons	0.78	0.78	0.78	0.78	0.79	0.79	0.79	0.79
3	0.0001	32	1 hidden layer with 16 neurons	0.74	0.73	0.73	0.74	0.75	0.75	0.75	0.75
4	0.0001	64	1 hidden layer with 16 neurons	0.74	0.74	0.74	0.74	0.74	0.74	0.74	0.74
5	0.0001	32	2 hidden layers with 32 and 16 neurons respectively	0.71	0.71	0.71	0.71	0.72	0.72	0.72	0.72
6	0.0001	64	2 hidden layers with 32 and 16 neurons respectively	0.73	0.73	0.73	0.73	0.72	0.73	0.73	0.73
7	0.01	32	1 hidden layers with 32	0.79	0.79	0.79	0.79	0.79	0.79	0.79	0.79
8	0.01	64	1 hidden layers with 32	0.78	0.78	0.78	0.78	0.79	0.79	0.79	0.79

Table 1: Table containing the training and validation results obtained during hyperparameter tuning

8 Test and Results

To evaluate the results of applying the model on unseen data, several evaluation metrics were used. In addition to the classic ones such as precision, recall and accuracy, metrics such as f1-score, confusion matrix and roc-auc score were also used. In particular, a confusion matrix is a table that summarizes the performance of a classification model by comparing its predicted labels to the true labels. It displays the number of true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN) of the model's predictions.
Accuracy is the number of samples correctly classified out of all the samples present in the test set:

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN}$$

Precision is the number of samples actually belonging to the positive class out of all the samples that were predicted to be of the positive class by the model:

$$Precision = \frac{TP}{TP + FP}$$

Recall is the number of samples predicted correctly to be belonging to the positive class out of all the samples that actually belong to the positive class:

$$Recall = \frac{TP}{TP + FN}$$

F1-Score is the harmonic mean of the precision and recall scores obtained for the positive class:

$$F1 - Score = \frac{2xPrecisionxRecall}{Precision + Recall}$$

ROC AUC score shows how well the classifier distinguishes positive and negative classes. It can take values from 0 to 1. A higher ROC AUC indicates better performance. A perfect model would have an AUC of 1, while a random model would have an AUC of 0.5.
The results obtained on the test set using the model with the parameters with which the best results were obtained on the training and validation set will be shown below. As we can see in the table 1, the best results were obtained using a learning rate of 0.01 and 0.0001 and a batch size of 32. Therefore it was decided to observe the results obtained on the test set in both cases in order to be able to choose the best between the two.
In tables 2 and 3 it is possible to observe the results obtained for the precision, recall, f1-score and accuracy metrics for each of the classes. It can be seen that the values obtained for class 0.0 and class 4.0 (corresponding to ratings of 1.0 and 5.0) are quite good, while the model tends to have greater difficulties in classifying texts relating to classes 1.0, 2.0 and 3.0 (ratings 2.0, 3.0 and 4.0 respectively). Furthermore, it can be observed that the values in the two tables are almost the same.

	precision	recall	f1-score	support
0.0	0.84	0.85	0.85	2621
1.0	0.77	0.76	0.76	2610
2.0	0.75	0.75	0.75	2559
3.0	0.77	0.75	0.76	2611
4.0	0.83	0.85	0.84	2594
macro avg	0.79	0.79	0.79	12995
weighted avg	0.79	0.79	0.79	12995
accuracy			0.79	

Table 2: Table containing the evaluation metrics on the test set for model with learning-rate = 0.01 and batch-size = 32

	precision	recall	f1-score	support
0.0	0.84	0.85	0.85	2621
1.0	0.77	0.76	0.77	2612
2.0	0.76	0.76	0.76	2557
3.0	0.76	0.76	0.76	2612
4.0	0.82	0.84	0.83	2594
macro avg	0.79	0.79	0.79	12997
weighted avg	0.79	0.79	0.79	12997
accuracy			0.79	

Table 3: Table containing the evaluation metrics on the test set for model with learning-rate = 0.0001 and batch-size = 32

As regards the computation of the roc-auc score, the one-vs-all approach was used which involves splitting the multi-class dataset into multiple binary classification problems. A binary classifier is then trained on each binary classification problem and predictions are made using the model that is the most confident. In particular, the code suggested by Kaggle was used. Tables 4 and 5 show the roc-auc scores for the various classes. Also in this case we can see how the values obtained are practically the same. These results show us how the model, using these parameters, has a good capability to distinguish between classes.

class	roc-auc score
0.0	0.90
1.0	0.85
2.0	0.84
3.0	0.84
4.0	0.90

Table 4: Table containing the roc-auc score on the test set for model with learning-rate = 0.01 and batch-size = 32

class	roc-auc score
0.0	0.90
1.0	0.85
2.0	0.84
3.0	0.84
4.0	0.90

Table 5: Table containing the roc-auc score on the test set for model with learning-rate = 0.0001 and batch-size = 32

For what concern the confusion matrix, it was created using the MulticlassMetrics class available in the Spark ML library. In particular, on the diagonal of the matrix we will have the samples that have been classified correctly, while the cells outside the diagonal show the model errors. In figures 1 and 2 it is possible to observe the results produced by the use of the confusion matrix. In particular, looking at figure 1, you can see that despite having obtained good results in the other metrics used for class 0.0, in this case it can be noted that reviews belonging to this class are often incorrectly labeled by the model as belonging to class 1.0. The same thing happens for class 1.0 reviews. In fact, these are often erroneously predicted by the model as belonging to classes 0.0 or 2.0.
As regards the results visible in figure 2, it can be seen that the results produced by the confusion matrix are better in the two classes we discussed above, especially in the class 1.0. In fact, 500 to 600 more reviews are correctly classified than before. The results obtained for the other classes, however, appear to be rather similar in the two confusion matrices.
What is observable from both confusion matrices is the fact that the main classification errors concern a misclassification with neighboring classes. In fact, we can observe how the greatest number of incorrectly classified reviews are found in correspondence with the diagonal values.
Based on these presented results, it was decided to maintain the model with a learning rate equal to 0.0001 and batch size equal to 32 for reproducing the experiment.

9 Conclusion

Regarding the results obtained from the classification, they can be considered quite good. However, we can observe that the TF.IDF method for extracting features is perhaps not the most suitable. The main reason is that the TF.IDF is a measure that gives importance to individual words that may represent the characterizing words of a text or document, but does not capture the context. This can lead to obtaining better results as for example with regard to the 4.0 class (corresponding to the 5.0 rating) because the corresponding reviews could contain words that have a very positive meaning (referring to the fact that they are very good reviews) that do not appear in the reviews with different ratings values and consequently this could help in classification.

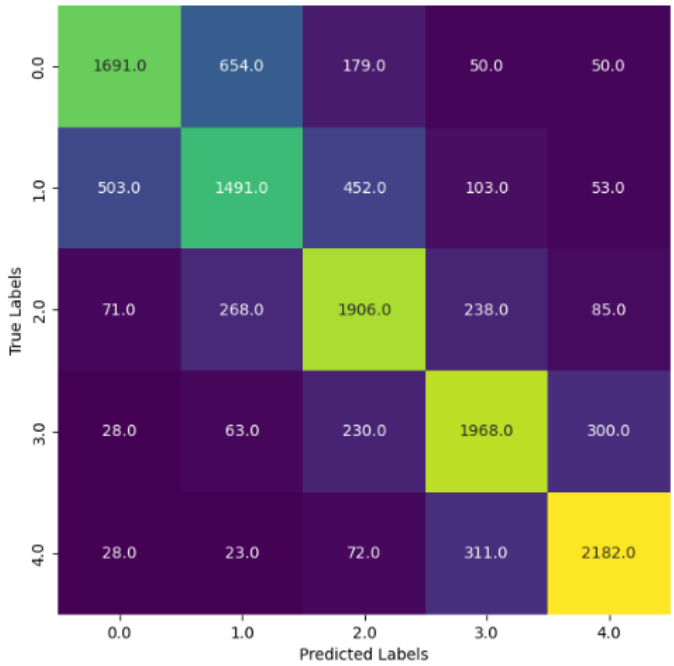

Fig. 1: Confusion matrix obtained through the model with learning rate = 0.01 and batch size = 32

Probably better results would be obtained using models to extract features taking into account the context of the text.

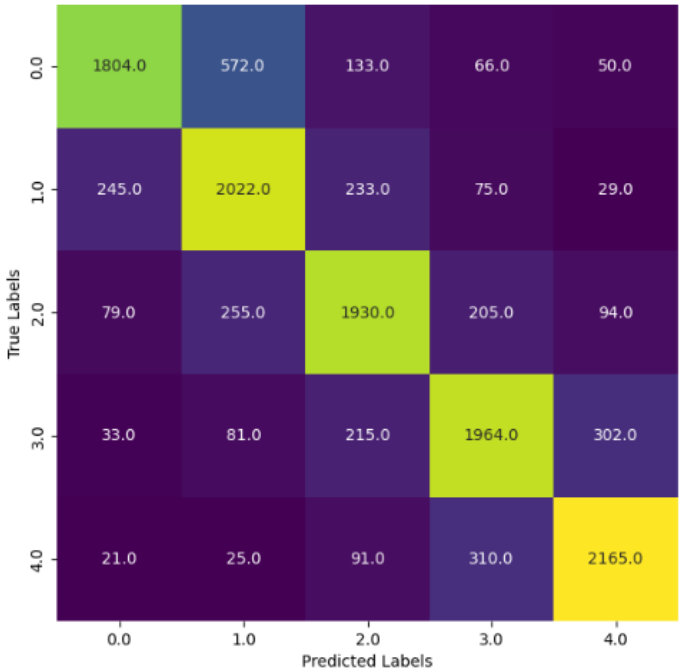

Fig. 2: Confusion matrix obtained through the model with learning rate = 0.0001 and batch size = 32